Lecture 20

Searching & Sorting

Linear Search

```
def linear_search(v,b):
    """Returns: first occurrence of v in b (-1 if not found)
    Precond: b a list of number, v a number
    """
    # Loop variable
    i = 0
    while i < len(b) and b[i] != v:
        i = i + 1

    if i == len(b): # not found
        return -1
    return i
```

How many entries do we have to look at?

Linear Search

```
def linear_search(v,b):
    """Returns: first occurrence of v in b (-1 if not found)
    Precond: b a list of number, v a number
    """
    # Loop variable
    i = 0
    while i < len(b) and b[i] != v:
        i = i + 1

    if i == len(b): # not found
        return -1
    return i
```

How many entries do we have to look at?

All of them!

Linear Search

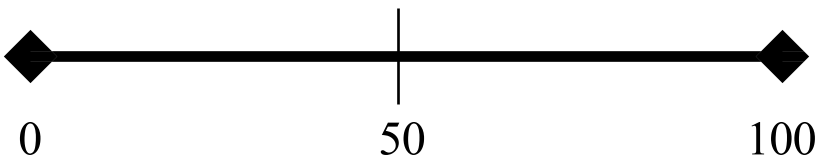
```
def linear_search(v,b):
    """Returns: last occurrence of v in b (-1 if not found)
    Precond: b a list of number, v a number
    """
    # Loop variable
    i = len(b)-1
    while i >= 0 and b[i] != v:
        i = i - 1

    # Equals -1 if not found
    return i
```

How many entries do we have to look at?

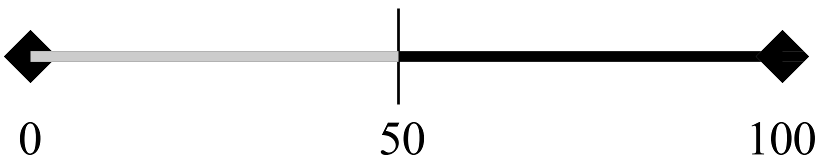
All of them!

Is There a Better Way?



- Thinking of number 0..100
  - You get to guess number
  - I tell you higher or lower
  - Continue until get it right
- **Goal:** Keep # guesses low
  - Use my answers to help
- **Strategy?**
  - Start guess in the middle
  - Answer eliminates half
  - Go to middle of remaining

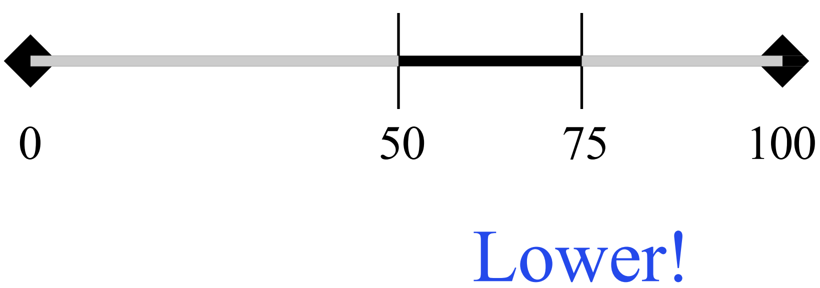
Is There a Better Way?



Higher!

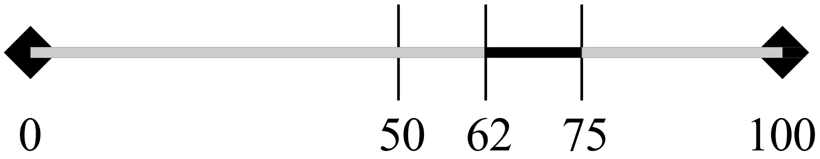
- Thinking of number 0..100
  - You get to guess number
  - I tell you higher or lower
  - Continue until get it right
- **Goal:** Keep # guesses low
  - Use my answers to help
- **Strategy?**
  - Start guess in the middle
  - Answer eliminates half
  - Go to middle of remaining

Is There a Better Way?



- Thinking of number 0..100
  - You get to guess number
  - I tell you higher or lower
  - Continue until get it right
- **Goal:** Keep # guesses low
  - Use my answers to help
- **Strategy?**
  - Start guess in the middle
  - Answer eliminates half
  - Go to middle of remaining

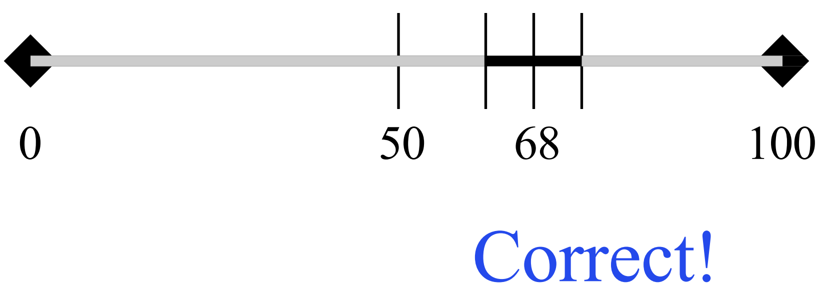
Is There a Better Way?



Higher!

- Thinking of number 0..100
  - You get to guess number
  - I tell you higher or lower
  - Continue until get it right
- **Goal:** Keep # guesses low
  - Use my answers to help
- **Strategy?**
  - Start guess in the middle
  - Answer eliminates half
  - Go to middle of remaining

Is There a Better Way?



- Thinking of number 0..100
  - You get to guess number
  - I tell you higher or lower
  - Continue until get it right
- **Goal:** Keep # guesses low
  - Use my answers to help
- **Strategy?**
  - Start guess in the middle
  - Answer eliminates half
  - Go to middle of remaining

Binary Search

```
def binary_search(v,b):
    # Loop variable(s)
    i = 0, j = len(b) - 1
    while i <= j:
        mid = (i+j)//2
```

Binary Search

```
def binary_search(v,b):
    # Loop variable(s)
    i = 0, j = len(b) - 1
    while i <= j:
        mid = (i+j)//2
        if b[mid] < v:
            i = mid+1
        elif b[mid] > v:
            j = mid-1
        else:
            return mid
    return -1 # not found
```

Requires that the data is sorted!

But few checks!

Observation About Sorting

- Sorting data can speed up searching
  - Sorting takes time, but do it once
  - Afterwards, can search many times
- Not just searching. Also speeds up
  - Duplicate elimination in data sets
  - Data compression
  - Physics computations in computer games
- Why it is a major area of computer science

The Sorting Challenge

- **Given:** A list of numbers
- **Goal:** Sort those numbers using only
  - Iteration (while-loops or for-loops)
  - Comparisons (< or >)
  - Assignment statements
- Why? For proper **analysis**.
  - Methods/functions come with hidden costs
  - Everything above has no hidden costs
  - Each comparison or assignment is “1 step”

This Requires Some Notation

- As the list is sorted…
  - Part of the list **will** be sorted
  - Part of the list will **not** be sorted
- Need a way to refer to portions of the list
  - Notation to refer to sorted/unsorted parts
- And have to do it **without** slicing!
  - Slicing makes a **copy**
  - Want to sort original list, not a copy

This Requires Some Notation

- As the list is sorted…
  - Part of the list **will** be sorted
  - Part of the list will **not** be sorted
- Need a way to refer to portions of the list
  - Notation to refer to sorted/unsorted parts
- And have to do it **without** slicing!
  - Slicing makes a **copy**
  - Want to sort original list, not a copy

Recall: Range Notation

- m..n is a range containing n+1-m values
  - 2..5 contains 2, 3, 4, 5. Contains 5+1 – 2 = 4 values
  - 2..4 contains 2, 3, 4. Contains 4+1 – 2 = 3 values
  - 2..3 contains 2, 3. Contains 3+1 – 2 = 2 values
  - 2..2 contains 2. Contains 2+1 – 2 = 1 values
  - 2..1 contains ???
- The notation m..n, always implies that m <= n+1
  - So you can assume that even if we do not say it
  - If m = n+1, the range has 0 values

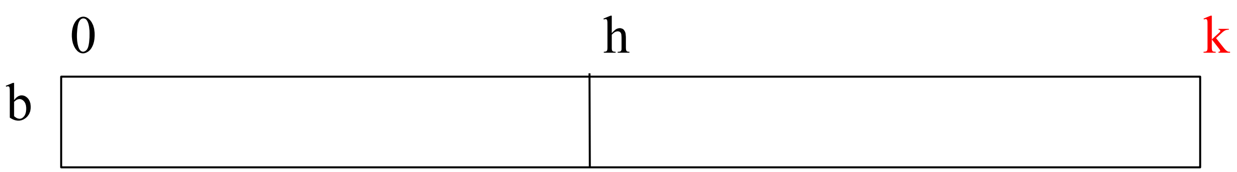
Recall: Range Notation

- m..n is a range containing n+1-m values
  - 2..5 contains 2, 3, 4, 5. Contains 5+1 – 2 = 4 values
  - 2..4 contains 2, 3, 4. Contains 4+1 – 2 = 3 values
  - 2..3 contains 2, 3. Contains 3+1 – 2 = 2 values
  - 2..2 contains 2. Contains 2+1 – 2 = 1 values
  - 2..1 contains ???

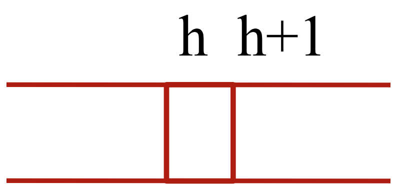
- The notation m..n, always implies that m <= n+1
  - So you can assume that even if we do not say it
  - If m = n+1, the range has 0 values

Horizontal Notation

- Want a pictoral way to visualize this sorting
  - Represent the list as long rectangle
  - We saw this idea in divide-and-conquer



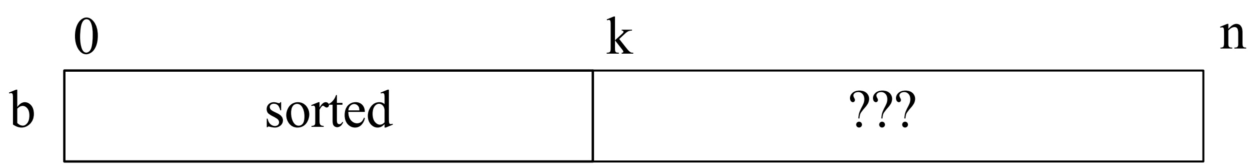
- Do **not** show individual boxes
  - Just dividing lines between regions
  - Label dividing lines with indices
  - But index is either left or right of dividing line



$(h+1) - h = 1$

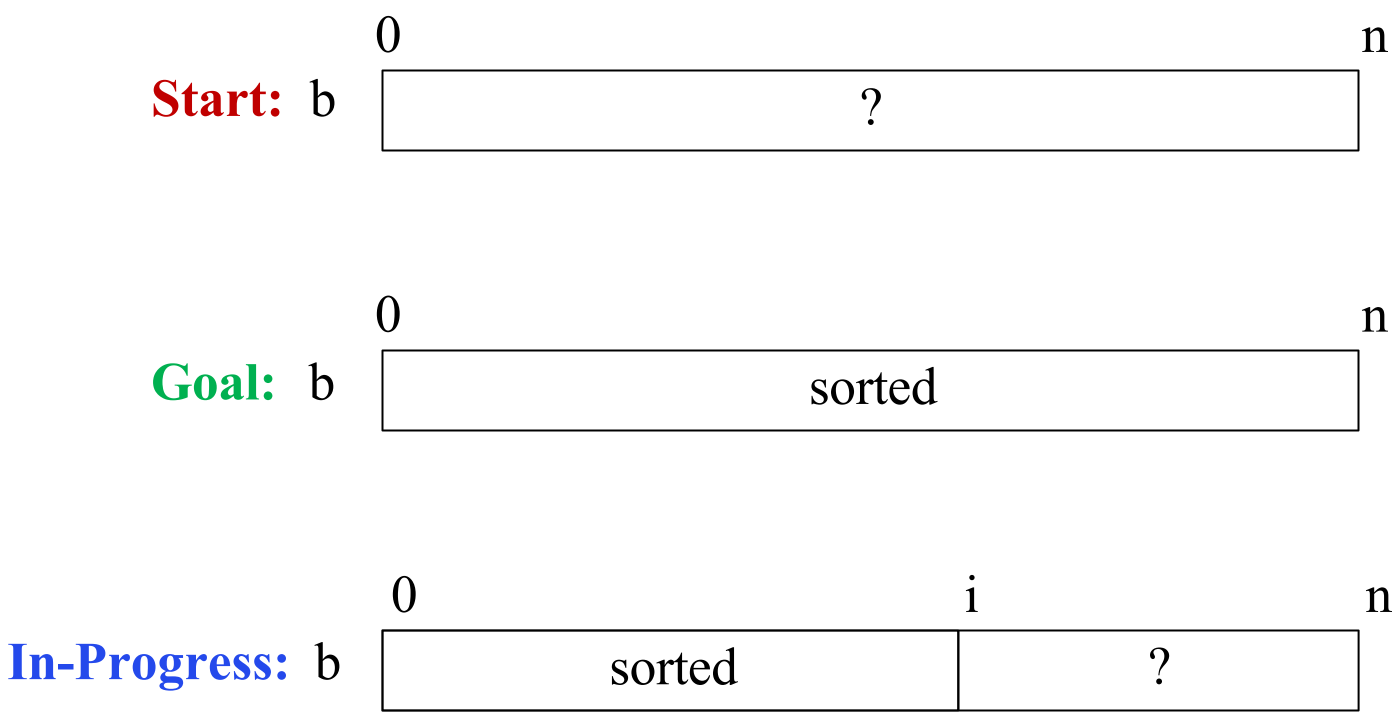
Horizontal Notation

- Label regions with properties
  - **Example:** Sorted or ???

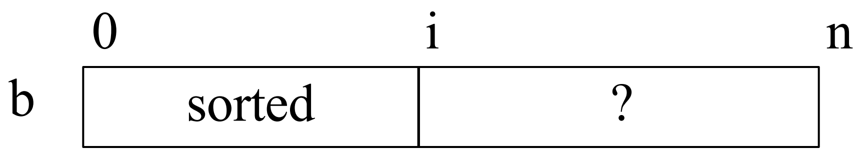


  - b[0..k–1] is sorted
  - b[k..n-1] **unknown** (might be sorted)
- Picture allows us to track progress

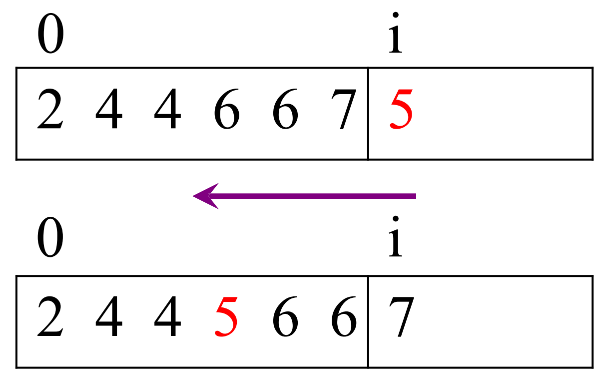
Visualizing Sorting



Insertion Sort




```
i = 0
while i < n:
    # Push b[i] down into its
    # sorted position in b[0..i]
    i = i+1
```

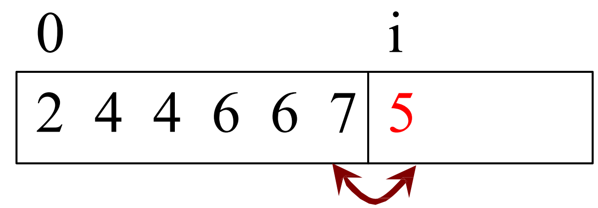


Remember the restrictions!

Insertion Sort: Moving into Position

```
i = 0
while i < n:
    push_down(b,i)
    i = i+1

def push_down(b, i):
    j = i
    while j > 0:
        if b[j-1] > b[j]:
            swap(b,j-1,j)
        j = j-1
```

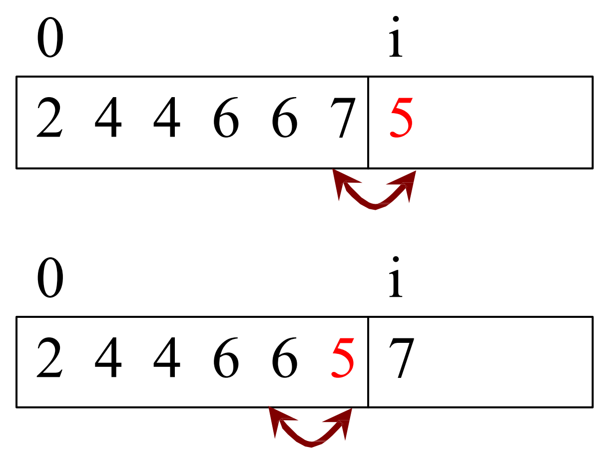


swap shown in the lecture about lists

Insertion Sort: Moving into Position

```
i = 0
while i < n:
    push_down(b,i)
    i = i+1

def push_down(b, i):
    j = i
    while j > 0:
        if b[j-1] > b[j]:
            swap(b,j-1,j)
        j = j-1
```



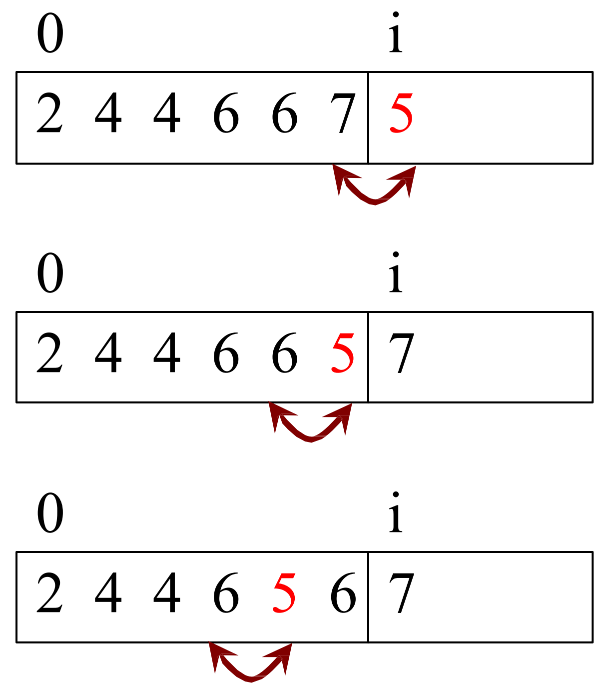
swap shown in the lecture about lists

Insertion Sort: Moving into Position

```
i = 0
while i < n:
    push_down(b,i)
    i = i+1

def push_down(b, i):
    j = i
    while j > 0:
        if b[j-1] > b[j]:
            swap(b,j-1,j)
        j = j-1
```

swap shown in the lecture about lists



Insertion Sort: Moving into Position

```
i = 0
while i < n:
    push_down(b,i)
    i = i+1

def push_down(b, i):
    j = i
    while j > 0:
        if b[j-1] > b[j]:
            swap(b,j-1,j)
        j = j-1
```

swap shown in the lecture about lists

0						i
2	4	4	6	6	7	5

0						i
2	4	4	6	6	5	7

0						i
2	4	4	6	5	6	7

0						i
2	4	4	5	6	6	7

The Importance of Helper Functions

```
i = 0
while i < n:
    push_down(b,i)
    i = i+1

def push_down(b, i):
    j = i
    while j > 0:
        if b[j-1] > b[j]:
            swap(b,j-1,j)
        j = j-1
```

VS

Can you understand all this code below?

```
i = 0
while i < n:
    j = i
    while j > 0:
        if b[j-1] > b[j]:
            temp = b[j]
            b[j] = b[j-1]
            b[j-1] = temp
        j = j -1
    i = i +1
```

Measuring Performance

- Performance is a tricky thing to measure
  - Different computers run at different speeds
  - Memory also has a major effect as well
- Need an independent way to measure
  - Measure in terms of “basic steps”
  - **Example:** Searching counted # of checks
- For sorting, we measure in terms of **swaps**
  - Three assignment statements
  - Present in all sorting algorithms

Insertion Sort: Performance

```
def push_down(b, i):
    """Push value at position i into
    sorted position in b[0..i-1]"""
    j = i
    while j > 0:
        if b[j-1] > b[j]:
            swap(b,j-1,j)
        j = j-1
```

- b[0..i-1]: i elements
- Worst case:
  - i = 0: 0 swaps
  - i = 1: 1 swap
  - i = 2: 2 swaps
- Pushdown is in a loop
  - Called for i in 0..n
  - i swaps each time

Total Swaps: $0 + 1 + 2 + 3 + \ldots (n-1) = (n-1)*n/2 = (n^2-n)/2$

Insertion Sort: Performance

```
def push_down(b, i):
    """Push value at position i into
    sorted position in b[0..i-1]"""
    j = i
    while j > 0:
        if b[j-1] > b[j]:
            swap(b,j-1,j)
        j = j-1
```

- b[0..i-1]: i elements
- Worst case:
  - i = 0: 0 swaps
  - i = 1: 1 swap
  - i = 2: 2 swaps
- Pushdown is in a loop
  - Called for i in 0..n
  - i swaps each time

Insertion sort is an n^2 algorithm

Total Swaps: $0 + 1 + 2 + 3 + \ldots (n-1) = (n-1)*n/2 = (n^2-n)/2$

Algorithm "Complexity"

- **Given**: a list of length n and a problem to solve
- **Complexity**: *rough* number of steps to solve worst case
- Suppose we can compute 1000 operations a second:

Complexity	n=10	n=100	n=1000
log n	0.003 s	0.006 s	0.01 s
n	0.01 s	0.1 s	1 s
n log n	0.016 s	0.32 s	4.79 s
n^2	0.1 s	10 s	16.7 m
n^3	1 s	16.7 m	11.6 d
2^n	1 s	4×10^{19} y	3×10^{290} y

Algorithm "Complexity"

- **Given**: a list of length n and a problem to solve
- **Complexity**: *rough* number of steps to solve worst case
- Suppose we can compute 1000 operations a second:

Complexity	n=10	n=100	n=1000
$\log n$	[illegible]	0.006 s	0.01 s
n	0.01 s	0.1 s	1 s
$n \log n$	0.016 s	0.32 s	4.79 s
n^2	0.1 s	10 s	16.7 m
n^3	1 s	16.7 m	11.6 d
2^n	1 s	$4\text{x}10^{19}$ y	$3\text{x}10^{290}$ y

Binary Search

Linear Search

Insertion Sort

Algorithm “Complexity”

- **Given**: a list of length n and a problem to solve
- **Complexity**: *rough* number of steps to solve worst case
- Suppose we can compute 1000 operations a second:

Complexity	n=10	n=100	n=1000
log n	0.003 s	0.006 s	0.01 s
n	0.0 s	0.1 s	1 s
n log n	0.016 s	0.32 s	4.79 s
n^2	0.1 s	10 s	16.7 m
n^3	1 s	16.7 m	11.6 d
2^n	1 s	4×10^{19} y	3×10^{290} y

Insertion Sort is Not Great

- Typically n^2 is okay, but not great
  - Will perform horribly on large data
  - Very bad when performance critical (games)
- We would like to do better than this
  - Can we get n swaps (**no**)?
  - How about n log n (**maybe**)
- This will require a new algorithm
  - Let's return to horizontal notation

A New Algorthm

		0	n
Start:	b	?	

		0	n
Goal:	b	sorted	

		0	i	n
In-Progress:	b	sorted, $\leq$ b[i..]	$\geq$ b[0..i-1]	

First segment always contains smaller values

Selection Sort

	0	i	n
b	sorted , ≤ b[i..]	≥ b[0..i-1]	

```
i = 0
while i < n:
    # Find minimum in b[i..]
    # Move it to position i
    i = i+1
```

	i	n
2 4 4 6 6	8 9 9 7 8 9	

	i	n
2 4 4 6 6	7 9 9 8 8 9	

	i	n
2 4 4 6 6 7	9 9 8 8 9	

Remember the restrictions!

Selection Sort

How fast is this?

```
i = 0
while i < n:
    j = index of min of b[i..n-1]
    swap(b,i,j)
    i = i+1
```

	i	n
2 4 4 6 6	8 9 9 7 8 9	

	i	n
2 4 4 6 6	7 9 9 8 8 9	

	i	n
2 4 4 6 6 7	9 9 8 8 9	

Selection Sort

This is also n^2!

```
i = 0
while i < n:
    j = index of min of b[i..n-1]
    swap(b,i,j)
    i = i+1
```

This is n steps

2	4	4	6	6	8 (i)	9	9	7	8	9 (n)
2	4	4	6	6	7 (i)	9	9	8	8	9 (n)
2	4	4	6	6	7	9 (i)	9	8	8	9 (n)

What is the Problem?

- Both insertion, selection sort are **nested loops**
  - **Outer loop** over each element to sort
  - **Inner loop** to put next element in place
  - Each loop is n steps. $n \times n = n^2$
- To do better we must *eliminate* a loop
  - But how do we do that?
  - What is like a loop? **Recursion!**
  - Will see how to do this next lecture